Изоляция

2.1. Согласованность

Важная особенность реляционных СУБД — обеспечение *согласованности* (consistency), то есть *корректности* данных.

Известно, что на уровне базы данных можно создавать *ограничения целостности* (integrity constraints), такие как `NOT NULL` или `UNIQUE`. СУБД следит за тем, чтобы данные никогда не нарушали эти ограничения, то есть оставались целостными.

Если бы все ограничения были сформулированы на уровне базы данных, согласованность была бы гарантирована. Но некоторые условия слишком сложны для этого, например охватывают сразу несколько таблиц. И даже если ограничение в принципе можно было бы определить в базе данных, но из каких-то соображений оно не определено, это не означает, что его можно нарушать.

Итак, согласованность строже, чем целостность, но что конкретно под ней понимается, СУБД не знает. Если приложение нарушит согласованность, не нарушая целостности, у СУБД не будет способа узнать об этом. Получается, что гарантом согласованности выступает приложение, и остается верить, что оно написано корректно и никогда не ошибается.

Но если приложение выполняет только корректные последовательности операторов, в чем тогда роль СУБД?

Во-первых, корректная последовательность операторов может временно нарушать согласованность данных, и это, как ни странно, нормально.

Заезженный, но понятный пример состоит в переводе средств с одного счета на другой. Правило согласованности может звучать так: *перевод никогда не меняет общей суммы денег на счетах*. Такое правило довольно трудно (хотя и возможно) записать на SQL в виде ограничения целостности, так что пусть оно существует на уровне приложения и остается невидимым для СУБД. Перевод состоит из двух операций: первая уменьшает средства на одном счете, вторая — увеличивает на другом. Первая операция нарушает согласованность данных, вторая — восстанавливает.

Если первая операция выполнится, а вторая — по причине какого-то сбоя — нет, то согласованность нарушится. А это недопустимо. Ценой неимоверных усилий такие ситуации можно обрабатывать на уровне приложения, но, к счастью, это не требуется. Задачу полностью решает СУБД, если знает, что две операции составляют неделимое целое, то есть *транзакцию*.

Но есть и второй, более тонкий момент. Транзакции, абсолютно правильные сами по себе, при одновременном выполнении могут начать работать некорректно. Это происходит из-за того, что перемешивается порядок выполнения операций разных транзакций. Если бы СУБД сначала выполняла все операции одной транзакции, а только потом — все операции другой, такой проблемы не возникало бы, но без распараллеливания работы производительность была бы невообразимо низкой.

Ситуации, когда корректные транзакции некорректно работают вместе, называются *аномалиями* одновременного выполнения.

Простой пример: если приложение хочет получить из базы согласованные данные, то оно как минимум не должно видеть изменения других незафиксированных транзакций. Иначе (если какая-либо транзакция будет отменена) можно увидеть состояние, в котором база данных никогда не находилась. Такая аномалия называется *грязным чтением*. Есть множество других, более сложных аномалий.

Роль СУБД состоит в том, чтобы выполнять транзакции параллельно и при этом гарантировать, что результат такого одновременного выполнения будет совпадать с результатом одного из возможных последовательных выполнений. Иными словами — *изолировать* транзакции друг от друга, устранив любые возможные аномалии.

Таким образом, транзакцией называется множество операций, которые переводят базу данных из одного корректного состояния в другое корректное состояние (*согласованность*) при условии, что транзакция выполнена полностью (*атомарность*) и без помех со стороны других транзакций (*изоляция*). Это определение объединяет требования, стоящие за первыми тремя буквами акронима ACID: Atomicity, Consistency, Isolation. Они настолько тесно связаны друг с другом, что рассматривать их по отдельности просто нет смысла. На самом деле сложно отделить и требование долговечности (Durability), ведь при крахе системы в ней остаются изменения незафиксированных транзакций, а с ними приходится что-то делать, чтобы восстановить согласованность данных.

Получается, что СУБД помогает приложению поддерживать согласованность, учитывая состав транзакции, но не имея при этом понятия о подразумеваемых правилах согласованности.

К сожалению, реализация полной изоляции — технически сложная задача, сопряженная с уменьшением производительности системы. Поэтому на практике почти всегда применяется ослабленная изоляция, которая предотвращает некоторые, но не все аномалии. А это означает, что часть работы по обеспечению согласованности данных ложится на приложение. Именно поэтому очень важно понимать, какой уровень изоляции используется в системе, какие гарантии он дает, а какие — нет, и как в таких условиях писать корректный код.

2.2. Уровни изоляции и аномалии в стандарте SQL

Стандарт SQL описывает четыре уровня изоляции[1]. Эти уровни определяются перечислением аномалий, которые допускаются или не допускаются

при одновременном выполнении транзакций. Поэтому разговор об уровнях придется начать с аномалий.

Стоит иметь в виду, что стандарт — некое теоретическое построение, которое влияет на практику, но с которым практика в то же время сильно расходится. Поэтому все примеры здесь умозрительные. Они будут использовать операции над счетами клиентов: это довольно наглядно, хотя, надо признать, не имеет ни малейшего отношения к тому, как банковские операции устроены в действительности.

Интересно, что со стандартом SQL расходится и настоящая теория баз данных[1], развившаяся уже после того, как стандарт был принят, а практика успела уйти вперед.

Потерянное обновление

Аномалия *потерянного обновления* (lost update) возникает, когда две транзакции читают одну и ту же строку таблицы, затем одна транзакция обновляет эту строку, после чего вторая транзакция обновляет эту же строку, не учитывая изменений, сделанных первой транзакцией.

Например, две транзакции собираются увеличить сумму на одном и том же счете на 100 ₽. Первая транзакция читает текущее значение (1000 ₽), затем вторая транзакция читает то же самое значение. Первая транзакция увеличивает сумму (получается 1100 ₽) и записывает в базу это новое значение. Вторая транзакция поступает так же: получает те же 1100 ₽ и записывает их. В результате клиент потерял 100 ₽.

Потерянное обновление не допускается стандартом ни на одном уровне изоляции.

Грязное чтение и Read Uncommitted

Аномалия *грязного чтения* (dirty read) возникает, когда транзакция читает еще не зафиксированные изменения, сделанные другой транзакцией.

Например, первая транзакция переводит 100 ₽ на пустой счет клиента, но не фиксирует изменение. Другая транзакция читает состояние счета (обновленное, но не зафиксированное) и позволяет клиенту снять наличные — несмотря на то, что первая транзакция прерывается и отменяет свои изменения, так что никаких денег на счете клиента нет.

Грязное чтение допускается стандартом на уровне Read Uncommitted.

Неповторяющееся чтение и Read Committed

Аномалия *неповторяющегося чтения* (non-repeatable read) возникает, когда транзакция читает одну и ту же строку два раза, а в промежутке между чтениями вторая транзакция изменяет (или удаляет) эту строку и фиксирует изменения. Тогда первая транзакция получит разные результаты.

Например, пусть правило согласованности *запрещает отрицательные суммы на счетах клиентов*. Первая транзакция собирается уменьшить сумму на счете на 100 ₽. Она проверяет текущее значение, получает 1000 ₽ и решает, что уменьшение возможно. В это время вторая транзакция уменьшает сумму на счете до нуля и фиксирует изменения. Если бы теперь первая транзакция повторно проверила сумму, она получила бы 0 ₽ (но она уже приняла решение уменьшить значение, и счет «уходит в минус»).

Неповторяющееся чтение допускается стандартом на уровнях Read Uncommitted и Read Committed.

Фантомное чтение и Repeatable Read

Аномалия *фантомного чтения* (phantom read) возникает, когда одна транзакция два раза читает набор строк по одинаковому условию, а в промежутке между чтениями другая транзакция добавляет строки, удовлетворяющие этому условию, и фиксирует изменения. Тогда первая транзакция получит разные наборы строк.

Например, пусть правило согласованности *запрещает клиенту иметь более трех счетов*. Первая транзакция собирается открыть новый счет, проверяет их текущее количество (скажем, два) и решает, что открытие возможно. В это время вторая транзакция тоже открывает клиенту новый счет и фиксирует изменения. Если бы теперь первая транзакция перепроверила количество, она получила бы три (но она уже выполняет открытие еще одного счета, и у клиента их оказывается четыре).

Фантомное чтение допускается стандартом на уровнях Read Uncommitted, Read Committed и Repeatable Read.

Отсутствие аномалий и Serializable

Стандарт определяет и уровень, на котором не допускаются никакие аномалии, — Serializable. И это совсем не то же самое, что запрет на потерянное обновление и на грязное, неповторяющееся и фантомное чтение. Дело в том, что существует значительно больше известных аномалий, чем перечислено в стандарте, и еще неизвестное число пока неизвестных.

Уровень Serializable должен предотвращать *любые* аномалии. Это означает, что на таком уровне разработчику приложения не надо думать об изоляции. Если транзакции выполняют корректные последовательности операторов, работая в одиночку, данные останутся согласованными и при одновременной работе этих транзакций.

В качестве иллюстрации приведу известную всем таблицу из стандарта, к которой для ясности добавлен последний столбец:

	потерянные изменения	грязное чтение	неповторяющееся чтение	фантомное чтение	другие аномалии
Read Uncommitted	–	да	да	да	да
Read Committed	–	–	да	да	да
Repeatable Read	–	–	–	да	да
Serializable	–	–	–	–	–

Почему именно эти аномалии?

Почему из множества возможных аномалий в стандарте перечислены только несколько, и почему именно такие?

Достоверно этого, видимо, никто не знает. Но не исключено, что о других аномалиях во времена принятия первых версий стандарта просто не задумывались, поскольку теория заметно отставала от практики.

Кроме того, предполагалось, что изоляция должна быть построена на блокировках. Идея широко применявшегося *протокола двухфазного блокирования* (2PL) состоит в том, что в процессе выполнения транзакция блокирует затронутые строки, а при завершении — освобождает блокировки. Сильно упрощая: чем больше блокировок захватывает транзакция, тем лучше она изолирована от других транзакций. Но и тем сильнее страдает производительность системы, поскольку вместо совместной работы транзакции начинают выстраиваться в очередь за одними и теми же строками.

Как мне представляется, разница между стандартными уровнями изоляции в значительной степени объясняется как раз количеством необходимых для их реализации блокировок.

Если транзакция блокирует изменяемые строки для изменений, но не для чтения, получаем уровень Read Uncommitted с возможностью прочитать незафиксированные данные.

Если изменяемые строки блокируются и для чтения, и для изменений, получаем уровень Read Committed: незафиксированные данные прочитать нельзя, но при повторном обращении к строке можно получить другое значение (неповторяющееся чтение).

Если для всех операций блокируются и читаемые, и изменяемые строки, получаем уровень Repeatable Read: повторное чтение строки будет выдавать то же значение.

Но с Serializable проблема: невозможно заблокировать строку, которой еще нет. Из-за этого остается возможность фантомного чтения: другая транзакция может добавить строку, попадающую под условия выполненного ранее запроса, и эта строка окажется в повторной выборке.

Поэтому для полной изоляции обычных блокировок не хватает — нужно блокировать не строки, а условия (предикаты). Такие *предикатные* блокировки были предложены еще в 1976 году при работе над System R, но их практическая применимость ограничена достаточно простыми условиями, для которых понятно, как объединять два разных предиката. До реализации предикатных блокировок в задуманном виде дело, насколько мне известно, не дошло ни в одной системе.